第1章

システムLSI設計，「演算回路」で差を付ける！

——実装段階での問題点を見極めて，高性能な回路を作る

森岡澄夫

システムLSI（SOC：system on a chip）を開発する上で，演算回路設計は避けて通れません．その理由は二つあります．一つは，ディジタル家電など，システムLSIが重点的に使われる機器では，画像処理や音声処理，エラー訂正処理などが数多く行われるからです．これらの処理には複雑な算術演算が使われます．もう一つは，実用レベルの演算回路では，処理がどんどん複雑かつ高性能になってきているからです．実用的な性能を引き出すには「ハードウェア化に適した演算アルゴリズム」などを使う必要があります．本稿では，数式を回路に実装する際の問題点を説明します．（著者）

「演算回路設計」と聞いて思い浮かべる対象は，人によってさまざまでしょう．従来は，加算器や乗算器といった「演算器」の論理設計（ゲート・レベル設計）を指す場合が多かったのではないかと思います．また，論理合成などのEDAツールが進歩した昨今では，「そのような知識はもう不要ではないか」というのも，一つの考え方です[注1]．

1 今こそ「演算回路設計」を学ぶとき

しかし現在，多くの演算が組み合わされた高度な処理をどうやって回路化するか，というテーマに対する関心がかつてないほど高まっています．そのような回路をEDAツール（設計自動化ツール）に100％任せきって自動設計するようなことはできず，さまざまな人知が必要です．すなわち，演算回路設計は決して過去の話などではなく，今も将来もホットであり続ける「ディジタル回路設計の要」だと言えます．

本稿では，ある程度まとまった機能を持つ演算回路，つまりデータ処理IPコアを対象とし，その設計にあたっての問題点やベースとなる考え方を説明します．もっとも，演算回路設計は今になって突然必要になってきたわけではありません（**図1**）．昔はこれが「システム設計」と呼ばれており，今と同じように重要な事柄でした．違うのは，昔はリーダ格のエンジニアだけが考えていれば済んだのに，今はずっと多くのエンジニアがかかわらざるを得なくなったことです．

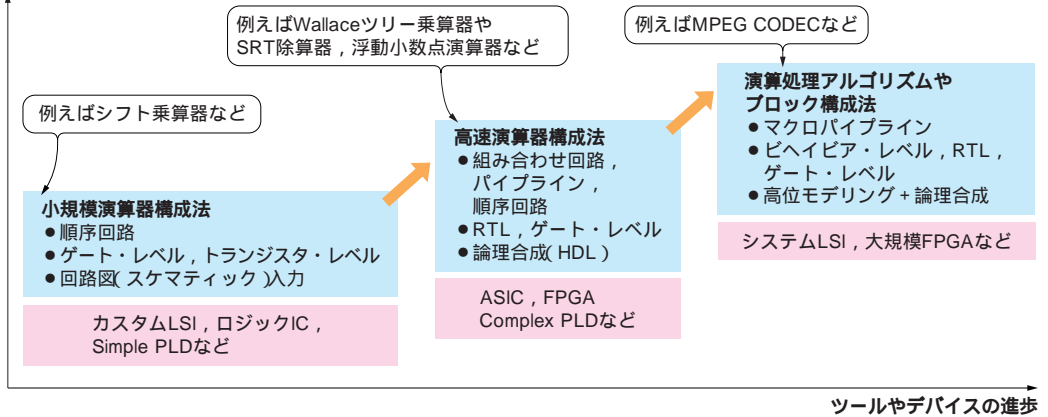
それでは，演算回路設計を知る必要性について，もう少し詳しく説明していきます．

● 必要性1：ほとんどのSOCに組み込まれている

まず，演算回路は現在設計されているシステムLSIのほとんどに組み込まれているからです．システムLSIの代表的な用途の一つに家電製品があります．例えば，携帯電話やデジタル・カメラ，DVDレコーダといった製品の機能を思い出せばすぐに想像できる通り，画像処理（2次元/3次

注1：筆者は，実設計では加算器や乗算器を“＋”，“×”となるべく高い抽象度で書き，ゲート・レベルで記述する（あるいはハード・コードする）のは必要な場所に限る，というスタンスをとっている．しかし，「演算器の論理設計の知識はもう不要だ」とは今のところ考えていない．RTL（register transfer level）とゲート・レベルでは，回路構造の考え方はかなり共通しているし，ツールでサポートされていない関数を作ろうとすると（こういうケースは頻繁にある），どうしても論理設計の知識が必要になる．

KeyWord　システムLSI，SOC，演算回路，ゲート・レベル，RTL，ビヘイビア・レベル，DSP，牧本ウェーブ，リードソロモン符号，RSA暗号，FFT

図1 演算回路設計において注目が集まるテーマの移り変わり
デバイスおよびEDAツールの能力向上によって，世間の関心は上位設計へ移っている．しかし，こと演算回路設計については，アルゴリズムや回路ブロック構成の重要性は今に始まったことではないし，ゲート・レベルでの演算器設計も不要になったわけではない．

元グラフィックス処理)や音声処理，データ圧縮・伸張処理，エラー訂正処理，暗号処理などが製品内部のシステムLSIには搭載されています(図2)．

いうまでもなく，こうした処理を行う回路では専門性の高い算術演算が実行されます．例えば，行列計算，データ領域変換，相関計算，距離計算など複雑な計算が行われます．さらにその内部では整数の四則演算はもちろん，多ビット長演算，固定小数点演算，浮動小数点演算，あるいは特殊な数体系上での演算が数多く使われます．

● 必要性２：自前で作らなければならない場合が多い

次に，こうしたシステムLSIで使われる演算回路は，しばしば自前で作らなければならないからです．

「通常，システムLSIにはCPUが搭載されているのだから，それほど複雑な処理が必要ならば，わざわざハードウェアを作らずに，ソフトウェアで処理すればよいのではないか？」という設計思想は，もちろん，間違いではありません．実際にそのようなシステムも多くあります．

ただし家電製品などでは，高性能であることが要求される演算処理については，専用ハードウェアで実装する傾向が顕著です(本稿でいう「高性能」とは，高速ということに限らず，回路規模が小さいことや消費電力が低いといったことも指す)．例えば携帯機器のように，限られた動作周波数や電源容量のもとで高解像度の画像を処理しようとす

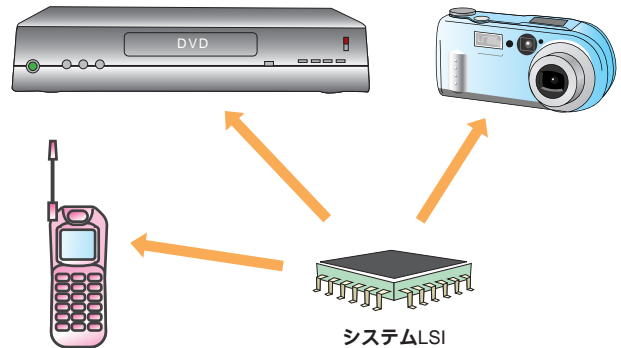
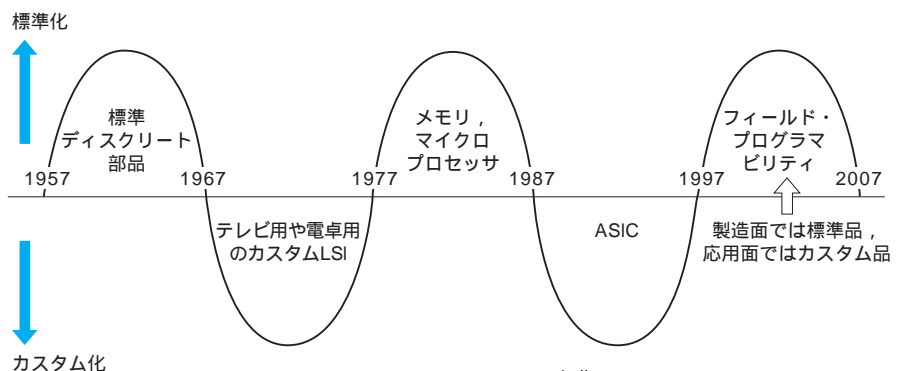



図2 ほとんどのシステムLSIで演算回路が使われる
システムLSIの代表的な用途は家電製品であるが，そこでは画像処理や音声処理などの高度なデータ処理が行われる．そのようなデータ処理は複雑な演算回路で実行される．

図3 時代が移り変わっても「高性能処理は専用ハードウェア」
半導体設計では，「専用ハードウェアを組むのが大変なのでプロセッサや共通プラットホームなどが流行する」→「設計手法やデバイスが進歩すると差異化のための専用ハードウェアが流行する」と，時代傾向が10年程度で循環している(図は有名な牧本ウェーブ)．しかし，「本当に高性能な処理はハードウェアで」という根底はずっと変わっていない．

れば，組み込み用途のプロセッサでは速度が足りない，あるいは電力を食い過ぎる，といった問題がしばしば起きてしまうからです(**図3**)．

なおかつ，自分のやりたい処理を実現できる市販のIPコアが存在しない場合が少なくありません．処理が規格などで明確に決まっているような演算であれば，自分でわざわざ回路を設計しなくても市販のIPコアで済む可能性はあります注2．ところが，画像処理など，システムLSIでよく行う演算処理では，そこに製品の差異化に直接結び付くような独自手法(例えば独自の画質向上，画像認識アルゴリズムなど)を用いることが多いのです．

● 必要性3：ツールでは良いアルゴリズムを作れない

さらに，実用演算回路を作る際に，設計を100％EDAツールに任せきるのは難しいからです．

もし，「$Y＝A×B＋C＋\cdots$」といった計算式から自動的に満足のいく回路ができてしまうのであれば，わざわざ演算回路の作り方を勉強する必要はないでしょう．

しかし，どのような演算処理でも100％ツールによる自動設計で済むようになる日は，近未来には来ないでしょう(**図4**)．製品の差異化に結び付くような実用レベルの演算処理回路の設計では，適切な処理アルゴリズムや回路ブロック構成を設計者が考えた上でツールを使わざるを得ないのです．これは，ソフトウェア作成においてCコンパイラがさまざまな最適化機能を持っていても，それがあくまで機械的な最適化の範ちゅうに留まり，悪いアルゴリズムを良いアルゴリズムにするような抜本的な最適化まではできないのと同じ事情です．もちろん，簡単な演算や頻繁に用いられる演算については，どんどんツールがサポートしていくはずですが，それにはおのずと限界があります．

また，特に実用演算回路の設計においては，ゲート・レベル設計(ANDやORといったゲート・レベル演算を使った設計)も長い間使われ続けていくことでしょう．本特集の第2章で例を示しますが，もしRTL(register transfer level)やビヘイビア・レベルといった高い抽象度の記述法だけしか使えないと，著しく冗長な回路ができてしまう場合が頻繁にあるからです．これは「演算回路はゲート・レベルやトランジスタ・レベルで設計しなければならない」といっているわけではありません．いろいろな抽象度を適切に使い分ける必要があるということです(1)．

図4　上手な設計と下手な設計の差が拡大する傾向にある

HDL(ハードウェア記述言語)や論理合成ツールが登場したころから「ハードウェアを意識しなくても誰でも設計できるようになる」と言われ続けているが，むしろ逆である．EDAツールが進歩しても，実用回路に対する要求水準がさらに高度化しており，ハードウェアを意識した設計とそうでない設計の差が以前よりも拡大する傾向にある．

2 演算処理の実装でぶつかる問題

演算処理の実装は決して単純労働などではなく，誰でも気軽にできるとは言い難いものです注3．また，工夫のしどころが多く，回路設計で最も面白いところです．

● 仕様の数式通り実装，では済まない

演算処理は仕様(内容)が数式で書かれていることが少なくありません．そのため，「仕様はきっちり決まっているし，プログラミング言語であれハードウェア記述言語であれ，数式を扱うのは得意なのだから，後は機械的に淡々と実装するだけだろう」とついつい想像してしまいます．

ところが，教科書などに書いてある例題と違い，実用レベルの演算処理では「仕様書に$y＝a＋b$と書かれていて，その通り言語で記述して設計が終わった」などというケースは少ないのです(筆者の経験ではほぼ皆無)．それがなぜかを以下に説明します．

● 数式が抽象的過ぎるという問題

やや極端な例ですが，仕様書に書かれた数式はたった1行，という場合があります．以下に有名な例をいくつか示します．

注2：もっとも，自分の作ろうとしているLSIに市販のIPコアがうまく適合しなかったり，またはIPコアにバグがあったりして，IPコアさえあれば気楽に設計できるかといえばそうでもないことはよく知られている．
注3：余談であるが，Cベース設計などが世間から注目を浴び続けている代表的な理由は，高度なデータ処理を回路化するニーズが高いのに，その設計がとても難しいからである．

● リードソロモン符号エンコーダ（エラー訂正）：

$$R(x) = M(x) \cdot x^{n-k} \bmod G(x) \quad \cdots\cdots (1)$$

● RSA暗号：

$$C = M^E \bmod N \quad \cdots\cdots (2)$$

● 高速フーリエ変換（FFT）：

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j(2\pi/N)nk} \quad \cdots\cdots (3)$$

いかがでしょうか．FFTがなんとなく難しそうですが，それは数式が込み入って見えるからです．数式を見ただけでは，これを回路にする上で一体何がどう難しいのかはダイレクトに伝わってこないので，「要するにこの式を計算すればいいんだな」と，簡単なことに思えるかもしれません．

ところが，いざこれらの演算回路を実際に作る側に立ってみると，なかなか一筋縄ではいきません（図5）．

図5　仕様書に書かれた演算処理を実装するために考えなければならないこと

演算処理回路は，その仕様が数式などで与えられるので，「ただその通り作ればよい，力仕事だろう」と勘違いしがち．しかし現実は全く異なり，数式の意味をつかんだ上で，アルゴリズムをはじめ多くの事項を回路設計者が補ってやらないと実用回路に仕上がらない．テストも相当厳しく，戦略的に行わないと，製品化できる水準にならない．回路設計の中でも最も難しい部類の仕事だが，それゆえに価値も高い．

最初にぶつかる難題は，具体的にどのような計算処理を組まなければならないのかを知るために，専門用語を勉強しながら数式を解読することです．例えば式（1）に示したリードソロモン符号の数式において，$M(x)$，$G(x)$，$R(x)$は関数に見えますが，そうではなくデータ系列（エラー訂正の分野では多項式と呼ぶ）です．しかも，データは整数ではなくて有限体(5)の数です．そうした初歩を踏まえた上で式（1）の意味を日本語で書き下してみると，

「長さkの有限体データの系列$M(x)$に$n-k$個のゼロ・データを付加し，その長さnのデータ系列を$n-1$次の多項式とみなし，有限体上の多項式$G(x)$で割って剰余$R(x)$を求め，それをデータ系列として出力する」

ということであって，決して，

「整数xを入力として，関数$M(x)$を計算してからx^{n-k}を掛けて，関数$G(x)$の計算結果で割る」

という意味ではありません．このように読み解くだけでもエラー訂正の世界の用語を調べねばならず，一苦労です．さらに，有限体とはどういうもので，有限体上の多項式の剰余とは一体何のことで，それはどのように計算するのかなどを一通り理解しないと，回路に落とし込める見通しすら立ちません．

つまり，仕様が簡単な数式だから実装も簡単かというと，“むしろ逆”であることが多いのです（単に数式の抽象度が非常に高いだけ注4）．数式上では，集合，行列，データ系列，整数など，何でも変数として扱えますし，どんなに複雑な処理を行う関数でも勝手に定義できます．また数式には，細かい計算手順（アルゴリズム）はほとんど何も表現されていません．このため，実装担当者が，数式と現物のハードウェアの間にある広くて深いギャップを埋めることになります注5．

● ツールが対応しないデータ型や関数を扱う際の問題

実際に演算処理を実装するとき，EDAツールがサポートしないデータ型や関数を扱わなければならない事態によく遭遇します．例えば，前述の式（1）～式（3）はすべてそれに該当しています．

- リードソロモン符号では有限体の多項式同士，および有限体の数同士の演算が必要
- RSA暗号では多ビット長の整数演算が必要．式（2）における変数M，E，Nは整数だが，実用では2,048ビットまたは1,024ビットの長さがある
- FFTでは複素数データを扱わなければならない

注4：「画像データをMPEG圧縮して暗号化し，エラー訂正をかける」といった猛烈に複雑な処理でも，画像データを変数A，圧縮を関数f，暗号化を関数g，エラー訂正を関数hとすれば，「$h(g(f(A)))$を計算する」とたった一言で仕様を表せてしまう．これは決して極端な例ではなく，MATLABやMathematicaなど高位モデリング・ツールは，これに近い非常に高い抽象度で処理をモデリングできる．そのおかげで，モデリング・ツール上で処理を考えるのは大変楽である．ところが，検討し終わった処理をいざC言語やHDLに落として製品化しようとすると，本稿で説明する問題に直面して四苦八苦することになる．

注5：結局のところ，この「仕様の理解が大変」という問題については，高性能な実用回路を開発する場合，地道に勉強するか人に相談するといった解決策しかなさそう．現在設計に苦しんでいる回路は，いずれIPコアがそろったりツールが整備されたりして楽に作れるようになるだろう．しかし，そのときにはまた何か別の高度な回路設計に苦しんでいるに違いない．魔法のような解決を期待している人にとっては残念かもしれないが…．

また，大半の画像処理や音声処理においては，整数ではなく小数で演算（浮動小数点演算や固定小数点演算）が行われますし，画像処理やエラー訂正処理などでは行列演算がしばしば必要になります．整数についても，必ずしも2の補数表現が使われるとは限りません．

ツールがサポートしていない演算を自分で組む必要が生じたために，仕様書のたった1行の数式が数千行，数万行ものソース・コードになってしまった，ということは珍しくありません．実際に，式(1)のリードソロモン符号や式(2)のRSA暗号なども，その程度の規模になり得ます．

● 演算精度を把握し，保証するという問題

画像処理や音声処理などの実装にあたっては，演算精度の保証やオーバフロー/アンダフローの発生に気を配ることが重要です（**図6**）．

仕様として数式が与えられたようなとき，入出力についてはビット数などが決められていたとしても，例えば次のようなことまでは「個々の実装の問題である」として指定されないのが普通です．

- 途中の計算を何ビットの精度で行えばよいのか
- 小数点以上/小数点以下に何ビットを持てばよいのか
- 有効けた数の調整では四捨五入するのか切り捨てるのか
- オーバフローやアンダフローが起きたときどうすればよいのか

このため，実装担当者がこれらの点を補うことになりますが，いい加減に決めてしまうと見つかりにくいバグの原因になる上，修正も容易ではないので注6，ソース・コードを書き始める以前での検討がかなり重要です．特に，チップ化するような回路の設計にあたっては，シミュレーションで場当たり的に決定するようなことをせず，きちんと理詰めで（証明を与えて）決定するようにします．

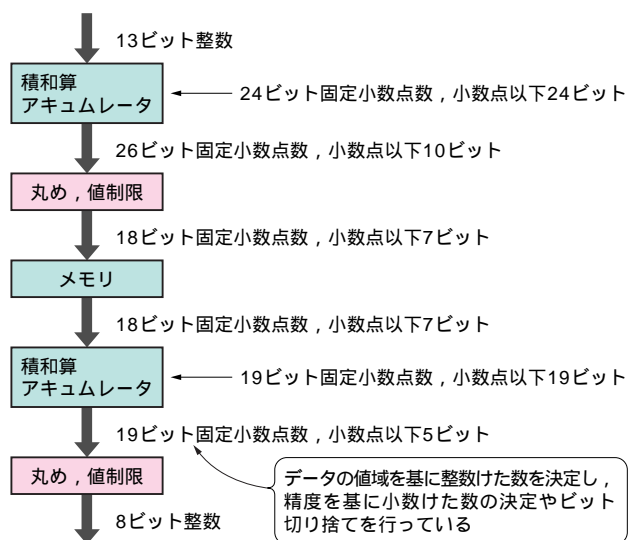



図6　演算回路設計では演算精度やビット長，オーバフローの管理がしばしば求められる

図は2次元逆離散コサイン変換（2D-IDCT）の例．画像処理や音声処理では，扱うデータが整数しかないような場合は少ない．通常は固定小数点演算が，場合によっては浮動小数点演算が使われる．処理の途中における数値ビット幅（ダイナミック・レンジ），有効けた数，オーバフローやアンダフローの有無，符号の有無などに相当注意を払わなければならない．

● 適切な回路処理性能を達成するという問題

これまでに述べた点は「仕様をハードウェアとして曲がりなりにも実装できるのか？」という問題だといえます．その次に出てくるのは，「目標とする回路性能を達成できるのか？」という問題です．例によって，仕様に書かれた数式は，どのような性能になるかは何も語ってくれません．

何も考えずに数式通りに計算すると痛い目に遭う，という一番面白い例が，先述のRSA暗号です（**図7**）．式(2)を素直に，

- MのE乗を求める
- それをNで割って余りを求める

と計算した場合を想像してみましょう．

M，E，Nが小さい数なら問題はなさそうです．ところが実用ではM，E，Nは例えば2,048ビットの数です．Eの最

注6：精度不足や丸め，オーバフロー処理に関係するバグは，回路を作って長時間に及ぶシミュレーションを実行した後に，なぜか時々画像が数箇所おかしくなる，といった形で問題が発覚しやすい．処理のどこに原因があるのかを特定する作業はかなり難しい．修正後は再びシミュレーション（リグレッション・テスト）にかける必要があり，そこで新たな問題が見つかって泥沼に陥ることがある．

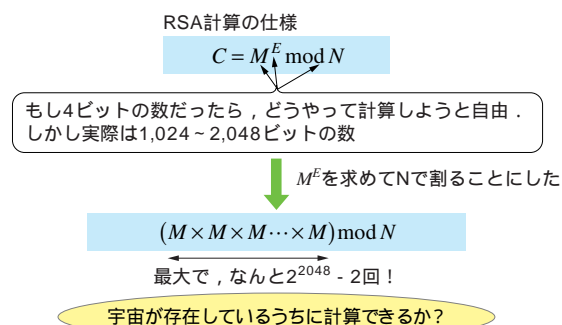



図7　数式通りに計算すると回路性能が悪化する場合

極端な例だが，RSA暗号の定義式をそのまま正直に計算すると，全く実用にならない．実用では入力が1,024ビット～2,048ビットと多ビットなので，計算時間が爆発してしまうからである．ここまでいかなくても，演算処理では，数式通り計算するのはあまりに非効率な場合が多い．

大値はなんと$2^{2048}-1$なので，Mを最大で$2^{2048}-2$回掛け合わせることになります．1GHzの動作周波数で1クロックに1回掛け合わせたとしても，$(2^{2048}-2)\div10^9$秒$>2^{2012}$秒$>2^{1987}$年はかかります．これでは宇宙が存在しているうちに計算が終わりません（筆者は物理学者ではないので，宇宙の存在年数は違うかもしれないが…）．また，2,048ビットのMを$2^{2048}-2$回も掛け合わせたら一体何ビットになるか，何mm^2のチップ・サイズになるか，という点についてもぜひ想像してみてください．いずれにしても，実装担当者がもっと勉強しなければならないのは明らかです（解決のヒントは参考文献(2)～(4)など）．

ここまで極端な例ではなくても，画像処理やデータ変換など，ほとんどの実用演算回路では，何も考えず数式通りに計算すると極端に性能が悪くなる場合があります．それは例えば，多量のデータ処理を行うソフトウェアを，アルゴリズムなどを何も考えずにコーディングした場合と同じ状況です．

● 数式で表現されない実装条件の問題

演算回路は，単に仕様通りの計算が実行できればよいのではなく，あくまでシステムに組み込まれて使われる部品です．そのため，次のようなIPコアとしての一般的な要件を満たす必要があります(6)．

- 入出力データをほかの回路やソフトウェアとうまくやりとりできること
- システム・バス上の適切な位置に組み込まれていること
- ユーザにとって使い勝手が良いこと
- レイアウトやチップ・テスト，（ほかの回路とつないだ上での）タイミング収束など，チップ化のための各作業，各工程で問題が起きないこと
- プログラマブル化を含め，できるだけ再利用性に優れていること

● 検証環境構築の問題

設計にばかり目が向いていると気が付きにくいのですが，相当ネックになるのが“作った回路の検証”です．演算回路の検証は，一般に難易度が高い上，時間もかかります（もちろん，チップ化できる水準まで厳しく検証する場合の話）．検証が大変になる理由は主に二つあります．

1）正しいテスト環境やテスト・ベクタを作ることが難しい

これまで「数式が抽象的過ぎて回路化できない」というような問題をいくつか挙げましたが，全く同じ理由でテスト環境構築やテスト・ベクタ生成もまた難しくなります．

MATLABのような上位モデリング・ツールが使えたり，製品に使った実績のあるソフトウェアやハードウェアがあるような場合，その結果と比較できるのでまだましです（「それらが100%正しいのか？」という問題は残るが…）．そのようなものがなければ，テスト・データ作成にも回路作成並みの労力がかかったり，回路動作とテスト・データのどちらが合っているのか分からないなど，なかなか困った事態が起こります（**図8(a)**）．

また，画像処理や科学技術計算などでは，回路出力値に一定の誤差を認めることがあります．このようなとき，テスト・データにもまた誤差があって，真の値が分からず，回路出力の誤差が許容範囲かどうかを判断できないこともあります．

そこで，ボトムアップにテストを積み上げる，複数のテスト環境で多数決する，あるいは設計・検証チームの体制・人数を見直すなど，ケース・バイ・ケースで検証戦略を考える必要があります．

2）どのようなテストをどこまですべきか判断が難しい

本質的には，すべての入力値を総当たりテストしないと，

(a) ゴールデン・データがどれか分からない，なぜ違うか分からない

(b) 全入力をテストできない

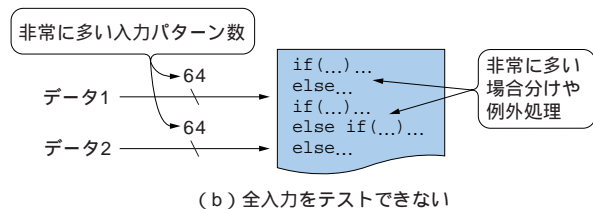
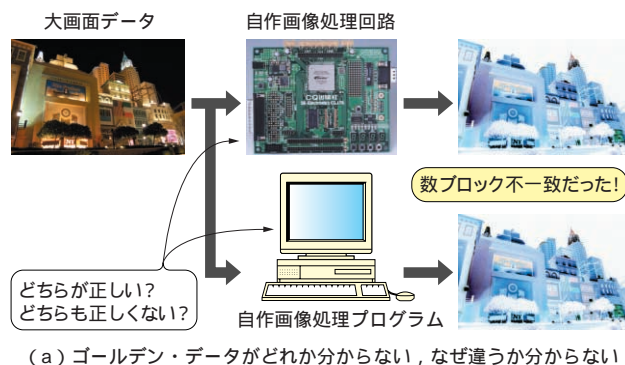



図8　検証では手法や環境，体制についての戦略が重要

実用演算回路の作成では，「だいたい正しそう」という試作水準まではあっという間に到達できても，「確実に問題がないと信じられる」というチップ化可能な水準に到達するには，かなりの時間がかかる．検証環境や人的体制をどう構築するかが鍵である．

設計した演算回路が「正しい」と判定できません注7(**図8(b)**)．非常に厳しいことを言うと，テストしていない入力値が残る演算回路をチップに搭載するのは，多かれ少なかれリスキーです(10年以上昔のことだが，米国Intel社のPentiumプロセッサで起きた浮動小数点除算のバグとその影響を思い出してほしい)．

現実には総当たりテストが不可能な場合がほとんどですが，コーナ・ケースを漏れなく押さえる，少しでも多くシミュレーションやFPGAなどを用いた長時間に及ぶ実機テストを行う，ホワイト・ボックス・テストや形式検証を併用するなど，ここでも検証についての戦略や人的体制の構築が重要です．なお，運良くチップが世に出る前にバグが見つかっても，演算回路をECO注8で修正するようなことは通常困難なので，とにかくフロントエンド設計(配置配線より前)の段階にあるうちに検証を完了させるのがベターです．

参考・引用*文献

(1) 森岡澄夫，高野光司，大庭信之；IPコアの設計時に立ちはだかる壁，Design Wave Magazine，2002年8月号，pp.120-126，CQ出版社．(設計時の抽象度の使い分け)
(2) 佐藤証，森岡澄夫；暗号処理のソフトv.s.ハード，Design Wave Magazine，2003年9月号，pp.72-79，CQ出版社．(RSA暗号の処理回路)
(3) 高野光司，大庭信之，森岡澄夫；C++を用いた公開かぎ暗号回路の細分化・階層化設計事例，Design Wave Magazine，2002年10月号，pp.148-157，CQ出版社．(C++からRSA暗号回路のRTLを導き出す)
(4) 高野光司，大庭信之，森岡澄夫；暗号回路の処理速度と設計効率を向上させるには，Design Wave Magazine，2002年12月号，pp.151-159，CQ出版社．(RSA暗号回路の設計トレードオフ調整など)
(5) 森岡澄夫；エラー訂正や暗号処理で使われる演算回路を極める，Design Wave Magazine，2003年7月号，pp.57-67，CQ出版社．(有限体の回路設計)
(6) 森岡澄夫；システム全体を見渡しながら回路設計を行う，Design Wave Magazine，2006年5月号，pp.49-63，CQ出版社．(ソフトウェアとハードウェアを合わせたシステム全体から見たIPコアの設計方法)

注7：コード・カバレッジを100%にすることとは違う．
注8：engineering change orderの略．レイアウトまで設計作業が完了した段階で，主に手作業で配線やセルを変更してロジックをデバッグすること．感覚としては「チップ製造のぎりぎり直前でのパッチ当て」であって，あまりたくさんの修正はできない．演算回路設計では，修正箇所は非常に多くなりやすい．

もりおか・すみお
日本電気(株)システムデバイス研究所

<筆者プロフィール>

森岡澄夫． NTT，IBM，ソニーの各社で，エラー訂正や暗号処理，画像処理などの高性能IPコア，システムLSIの研究開発を経験．代表作はPSP，PS3搭載セキュリティ・システム回路．現在は，多くの実開発経験をもとに，NEC・システムデバイス研究所にて動作合成システムCyberWorkBenchの研究開発に従事．筆者の子供(幼稚園児)は日本語を学習中だが，筆者はシステム・レベル設計言語を学習中．
http://www002.upp.so-net.ne.jp/morioka/

COLUMN 実用演算回路をテープアウトするまでの道のり1
～設計依頼からアルゴリズムの構築まで～

製品に搭載する演算回路(IPコア)を開発するときにどのような手順で作業が進むかについて，筆者がさまざまな会社やプロジェクトで経験した中での平均的な様子を紹介します．手順が若干前後したり，複数の手順がオーバラップして並行に進んだり，手戻ったりすることは日常茶飯事です．

手順1) IPコアの作成依頼が来る

依頼主は，上司や他部署，あるいは他会社などさまざまです．自分で「こういうチップを作ろう」と企画する場合もあるでしょう．

手順2) 作るべき処理の内容を理解する

正直なところ，筆者はここでいつも苦労しています．しかし，処理内容の勉強をさぼってハードウェアやソフトウェアの既製品を探し表面的に改造する，というやり方は演算処理用のIPコアでは望ましくありません．後工程で問題が起きたときの対処が困難だからです．また，利用した既製品にバグがあったとしても，その作成元に責任を押し付けることはできません(利用した自分が悪い)．

さらに，依頼主が処理の概要しか理解していないことが往々にしてあります．具体的に何を設計すべきかを逆に問い合わせて相談し，きっちり確認・合意しなければ受託者の責任になるので，自衛(言葉は悪いが)のためにも勉強をさぼることはできないのです．

手順3) システムへの回路の組み込み方を相談する

関係者とよく相談する必要があります．詳細については，本特集の第2章を参考にしてください．

手順4) 回路で用いるアルゴリズムや全体構造を考える

この工程は自分だけ，あるいは共同設計者との作業になります．なお，一つの回路ユニットの設計では，設計者1～3人+検証者1～2人くらいのチームとなることが多いようです(それ以上必要な場合，ユニットを分ける)．詳細は第2章を参照してください．

手順5) クリティカルな部分を検討し問題がないかチェックする

処理のうち回路の性能を決定付ける個所がどこかを予測し，理論的に検討したり回路を仮組みしたりして，重大な問題が起きないことを本格的な詳細設計の前に十分確認しておきます．

(第2章のコラム「実用演算回路をテープアウトするまでの道のり2」につづく)